CODEALPHA_SECURE_CODING_REVIEW

Name: Jahnavi Reddy Gowru

Reviewing a simple Node.js Express application

Introduction

The Node.js Express application under review serves as a foundational web platform for handling form submissions and file uploads. Built upon the Node.js runtime environment and the Express.js web framework, this application allows users to submit various data inputs, including their name, messages, and optionally, file attachments. Upon submission, the server processes the received data and provides users with confirmation messages, acknowledging their successful interactions with the platform.

While the application fulfills basic functionality requirements, it harbors several security vulnerabilities that necessitate attention.

This document provides a brief review of secure coding practices implemented in an application , Identifying potential security vulnerabilities and providing recommendations for secure coding practices.

Steps

- Choose a programming language and an application to review.
- Conduct a thorough review of code to identify security vulnerabilities.
- Provide recommendations for implementing secure coding practices.
- Utilise manual code reviews to assess the code.

Application Description

The application is a basic web application built using Node.js and Express.js, a popular web framework for Node.js. It allows users to submit a form containing their name, a message, and an optional file upload. Upon submission, the server processes the form data and displays a confirmation message to the user, including their name, message, and details of the uploaded file.

Node.js Express Application Code

```js
const express = require('express');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const app = express();
const upload = multer({ dest: 'uploads/' });
app.use(express.urlencoded({ extended: true }));
app.get('/', (req, res) => {
    res.send(`
        <form method="post" enctype="multipart/form-data" action="/upload">
            <label for="name">Name:</label>
            <input type="text" id="name" name="name"><br>
            <label for="message">Message:</label>
            <textarea id="message" name="message"></textarea><br>
            <label for="file">Upload a file:</label>
            <input type="file" id="file" name="file"><br>
            <button type="submit">Submit</button>
        </form>
    `);
});
app.post('/upload', upload.single('file'), (req, res) => {
    const name = req.body.name;
    const message = req.body.message;
    const file = req.file;

    // Process the form data
    res.send(`
        <h1>Hello, ${name}!</h1>
        <p>Your message: ${message}</p>
        <p>File uploaded successfully: ${file.originalname}</p>
    `);
});
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

Security Vulnerabilities and Recommendations

1) Cross-Site Scripting (XSS)

- **Issue**:
  The application echoes user input directly back to the browser without proper sanitization, leading to XSS vulnerabilities.
- **Recommendation**:
  Use a templating engine that automatically escapes output, such as EJS, Pug, or Handlebars. Alternatively, manually escape output using a library like escape-html

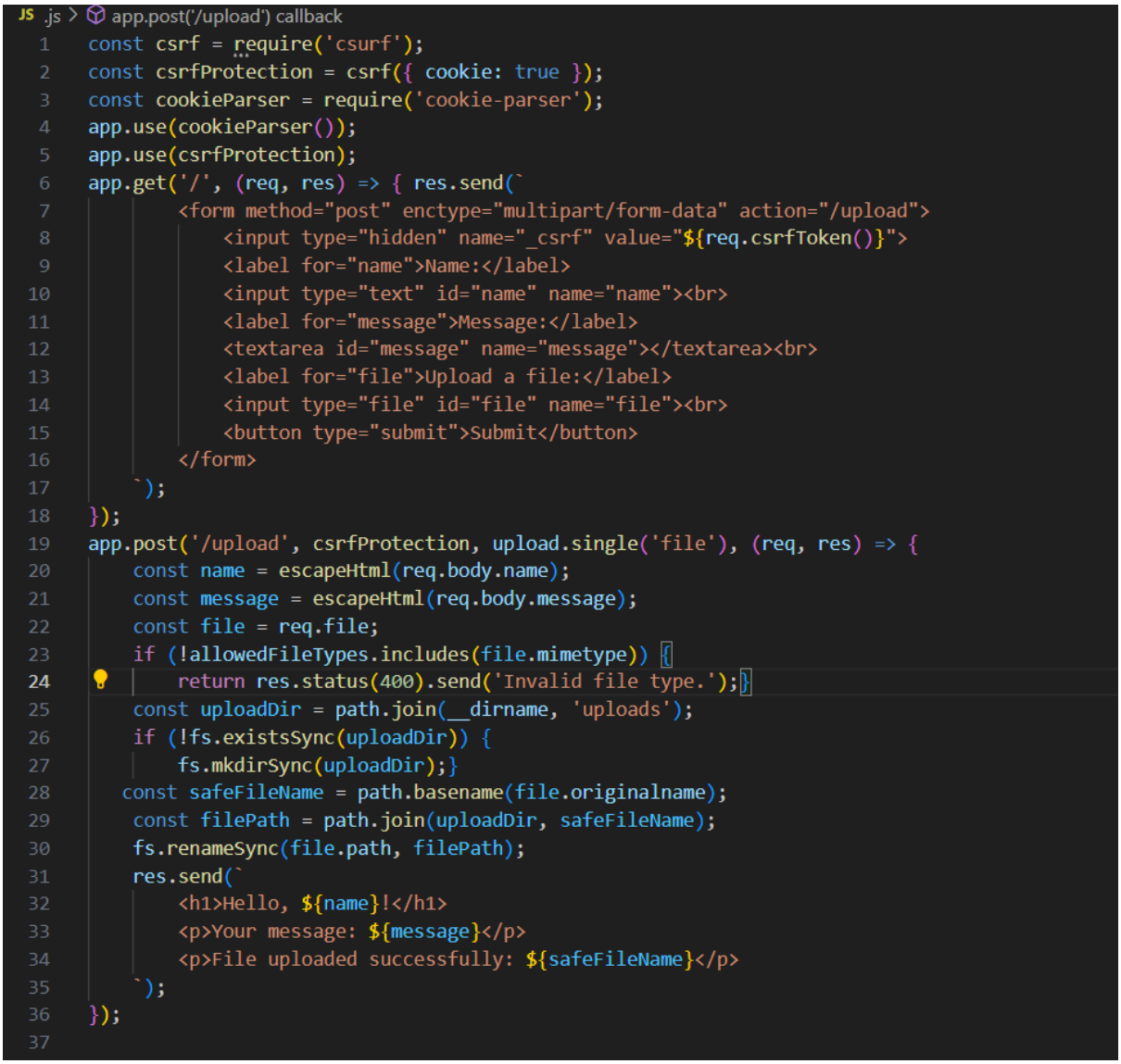
2)File Upload Vulnerability

- **Issue**:
  The application directly saves uploaded files to the uploads directory without validation, which can lead to directory traversal attacks and uploading of malicious files.
- **Recommendation**:
  Validate the uploaded file's name and type. Ensure the upload directory exists and is properly secured

```
.js > ...
const allowedFileTypes = ['image/jpeg', 'image/png', 'application/pdf'];

app.post('/upload', upload.single('file'), (req, res) => {
    const name = escapeHtml(req.body.name);
    const message = escapeHtml(req.body.message);
    const file = req.file;

    if (!allowedFileTypes.includes(file.mimetype)) {
        return res.status(400).send('Invalid file type.');
    }

    const uploadDir = path.join(__dirname, 'uploads');
    if (!fs.existsSync(uploadDir)) {
        fs.mkdirSync(uploadDir);
    }

    const safeFileName = path.basename(file.originalname);
    const filePath = path.join(uploadDir, safeFileName);
    fs.renameSync(file.path, filePath);

    res.send(`
        <h1>Hello, ${name}!</h1>
        <p>Your message: ${message}</p>
        <p>File uploaded successfully: ${safeFileName}</p>
    `);
});
```

3)Lack of CSRF Protection

- **Issue**:
  The form submission is vulnerable to Cross-Site Request Forgery (CSRF) attacks
- **Recommendation**:
  Implement CSRF tokens to protect against CSRF attacks using a library like ‘csurf’



```
JS .js > app.post('/upload') callback
const csrf = require('csurf');
const csrfProtection = csrf({ cookie: true });
const cookieParser = require('cookie-parser');
app.use(cookieParser());
app.use(csrfProtection);
app.get('/', (req, res) => { res.send(`
        <form method="post" enctype="multipart/form-data" action="/upload">
            <input type="hidden" name="_csrf" value="${req.csrfToken()}">
            <label for="name">Name:</label>
            <input type="text" id="name" name="name"><br>
            <label for="message">Message:</label>
            <textarea id="message" name="message"></textarea><br>
            <label for="file">Upload a file:</label>
            <input type="file" id="file" name="file"><br>
            <button type="submit">Submit</button>
        </form>
    `);
});
app.post('/upload', csrfProtection, upload.single('file'), (req, res) => {
    const name = escapeHtml(req.body.name);
    const message = escapeHtml(req.body.message);
    const file = req.file;
    if (!allowedFileTypes.includes(file.mimetype)) {
        return res.status(400).send('Invalid file type.');}
    const uploadDir = path.join(__dirname, 'uploads');
    if (!fs.existsSync(uploadDir)) {
        fs.mkdirSync(uploadDir);}
   const safeFileName = path.basename(file.originalname);
    const filePath = path.join(uploadDir, safeFileName);
    fs.renameSync(file.path, filePath);
    res.send(`
        <h1>Hello, ${name}!</h1>
        <p>Your message: ${message}</p>
        <p>File uploaded successfully: ${safeFileName}</p>
    `);
});
```

4) Error Handling

- **Issue**:
  The application does not handle potential errors during file upload properly, which can result in incomplete error messages to the user.
- **Recommendation**:
  Implement comprehensive error handling and logging.

```
app.post('/upload', upload.single('file'), (req, res) => {
    try {
        const name = escapeHtml(req.body.name);
        const message = escapeHtml(req.body.message);
        const file = req.file;

        if (!allowedFileTypes.includes(file.mimetype)) {
            return res.status(400).send('Invalid file type.');
        }

        const uploadDir = path.join(__dirname, 'uploads');
        if (!fs.existsSync(uploadDir)) {
            fs.mkdirSync(uploadDir);
        }

        const safeFileName = path.basename(file.originalname);
        const filePath = path.join(uploadDir, safeFileName);
        fs.renameSync(file.path, filePath);

        res.send(`
            <h1>Hello, ${name}!</h1>
            <p>Your message: ${message}</p>
            <p>File uploaded successfully: ${safeFileName}</p>
        `);
    } catch (err) {
        console.error('File upload error:', err);
        res.status(500).send('File upload failed.');
    }
});
```

5)Output Encoding

- **Issue**:
  The application does not use proper output encoding to prevent XSS attacks
- **Recommendation**:
  Use proper encoding libraries to encode user input before outputting it to the browser.

```
s .js > ...
1   const escapeHtml = require('escape-html');
2
3   app.post('/upload', upload.single('file'), (req, res) => {
4       const name = escapeHtml(req.body.name);
5       const message = escapeHtml(req.body.message);
6       const file = req.file;
7
8       res.send(`
9           <h1>Hello, ${name}!</h1>
10          <p>Your message: ${message}</p>
11          <p>File uploaded successfully: ${escapeHtml(file.originalname)}</p>
12      `);
13  });
14
```

Brief manual code review for the provided Node.js Express application:

1. **Input Validation and Sanitization**:

- The application lacks explicit input validation and sanitization for user inputs such as name and message. This could expose the application to security vulnerabilities such as XSS attacks or injection attacks.
- **Recommendation**: Implement input validation and sanitization using middleware like express-validator to ensure that user inputs are safe and conform to expected formats.

2. **File Upload Security**:

- The application handles file uploads using multer middleware but doesn't perform sufficient validation on the uploaded file. Lack of file type validation and content inspection could lead to security risks such as storing and executing malicious files.
- **Recommendation**: Validate file types and contents before storing them. Consider using libraries like file-type to detect MIME types and verify file integrity.

3. **CSRF Protection**:

- CSRF protection is not implemented in the application, making it vulnerable to CSRF attacks. Without CSRF tokens, malicious sites can forge requests on behalf of authenticated users, leading to unauthorized actions.
- **Recommendation**: Implement CSRF protection using middleware like csurf. Generate unique CSRF tokens for each user session and validate them on form submissions to prevent CSRF attacks.

4. **Error Handling**:

- Error handling is present but could be improved. The application catches errors during file upload but only logs them to the console without providing feedback to the user. Additionally, it returns a generic "File upload failed" message, lacking detail.
- **Recommendation**: Improve error handling by providing informative error messages to users and logging errors securely. Use appropriate HTTP status codes (e.g., 500 for internal server errors) along with error messages.

5. **Output Encoding**:

- The application doesn't perform output encoding to prevent XSS attacks when displaying user inputs back to the browser. This could allow attackers to inject malicious scripts into the HTML response.
- **Recommendation**: Use proper output encoding techniques (e.g., escape-html module) to sanitize user

inputs before rendering them in HTML responses, preventing XSS vulnerabilities.

6. **Dependency Security**:

- The application doesn't address potential security vulnerabilities in its dependencies. Regularly updating dependencies is crucial to mitigate known vulnerabilities.
- **Recommendation**: Set up dependency monitoring tools (e.g., npm audit) to identify and address security vulnerabilities in dependencies. Regularly update dependencies to the latest secure versions.

Conclusion

In conclusion, the provided Node.js Express application exhibits several security vulnerabilities that could potentially compromise the confidentiality, integrity, and availability of the system. These vulnerabilities include inadequate input validation and sanitization, lack of proper file upload security measures, absence of CSRF protection, suboptimal error handling, insufficient output encoding, absence of session management, and neglect of dependency security.

To address these vulnerabilities and enhance the overall security posture of the application, several remediation steps are recommended. These include implementing input validation and sanitization using middleware like express-validator, validating file uploads to prevent malicious file uploads, incorporating CSRF protection using middleware like csurf, improving error handling to provide informative error messages, performing output encoding to prevent XSS attacks, implementing secure session management using middleware like express-session, and regularly updating dependencies to mitigate known vulnerabilities.

By implementing these fixes and adopting secure coding practices, the application can better withstand common web security threats and provide a safer user experience. Additionally, ongoing security awareness, regular security audits, and proactive monitoring are essential to maintaining the security of the application in the face of evolving threats.